**Lab Manual– 8 – Multilayer perceptron**

An MLP is a type of neural network consisting of at least three layers: an input layer, one or more hidden layers, and an output layer. Each neuron in one layer is connected to every neuron in the next layer. The MLP learns by adjusting the weights of these connections to minimize the error of its predictions, using a process called backpropagation. It is widely used for solving complex problems like XOR classification, image recognition, and more.

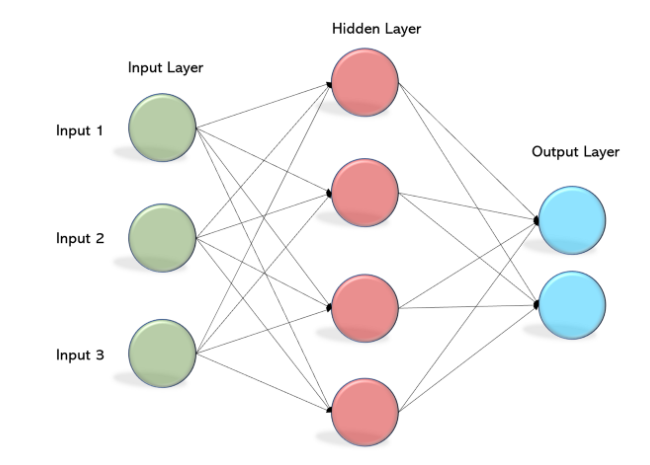


Figure 1. Multilayer Perceptron

The key components of the Multi-Layer Perceptron architecture are:

**Input Layer:**

* The initial layer, known as the input layer, receives the raw data or features.
* Each node in this layer represents a specific feature, forming the input vector.

**Hidden Layers:**

* Between the input and output layers, hidden layers process and transform the input data.
* Neurons within these layers apply weights and activation functions, allowing the network to capture intricate patterns and relationships.

**Output Layer:**

* The final layer, the output layer, produces the model’s predictions or classifications.
* The number of nodes in this layer depends on the nature of the task — one node for binary classification, multiple nodes for multi-class classification.

**Activation Functions:**

* Sigmoid
* ReLU
* Softmax

**Loss Function:**

* Cross-Entropy Loss: Measures the performance of a classification model.

**Optimization:**

* Gradient Descent: Updates weights to minimize the loss function.
* Backpropagation: Computes gradients of the loss function with respect to each weight.

**Step-by-Step Implementation**

1. Initialize Network Parameters

Initialize weights and biases: Randomly initialize weights and set biases to zero.

1. Forward Propagation

Compute hidden and output layer inputs and outputs: Apply activation functions to compute the activations.

1. Backward Propagation

Compute errors and update weights and biases: Adjust the weights and biases using the gradient descent algorithm.

1. Training the Model

Train the network: Perform forward and backward propagation for a specified number of epochs and print the loss periodically.

1. Predicting

Make predictions: Compute the output and return the predicted class labels.

**Sample Code:**

import numpy as np

import matplotlib.pyplot as plt

class MLP:

def \_\_init\_\_(self, input\_size, hidden\_size, output\_size, learning\_rate=0.1, epochs=10000):

self.input\_size = input\_size

self.hidden\_size = hidden\_size

self.output\_size = output\_size

self.learning\_rate = learning\_rate

self.epochs = epochs

# Initialize weights and biases

self.W1 = np.random.randn(self.hidden\_size, self.input\_size)

self.b1 = np.zeros((self.hidden\_size, 1))

self.W2 = np.random.randn(self.output\_size, self.hidden\_size)

self.b2 = np.zeros((self.output\_size, 1))

def sigmoid(self, x):

return 1 / (1 + np.exp(-x))

def sigmoid\_derivative(self, x):

return x \* (1 - x)

def forward(self, X):

"""Forward propagation"""

self.Z1 = np.dot(self.W1, X.T) + self.b1

self.A1 = self.sigmoid(self.Z1)

self.Z2 = np.dot(self.W2, self.A1) + self.b2

self.A2 = self.sigmoid(self.Z2)

return self.A2

def backward(self, X, y):

"""Backward propagation"""

m = X.shape[0]

y = y.reshape(1, m)

# Compute error

dZ2 = self.A2 - y

dW2 = (1 / m) \* np.dot(dZ2, self.A1.T)

db2 = (1 / m) \* np.sum(dZ2, axis=1, keepdims=True)

dZ1 = np.dot(self.W2.T, dZ2) \* self.sigmoid\_derivative(self.A1)

dW1 = (1 / m) \* np.dot(dZ1, X)

db1 = (1 / m) \* np.sum(dZ1, axis=1, keepdims=True)

# Update weights and biases

self.W1 -= self.learning\_rate \* dW1

self.b1 -= self.learning\_rate \* db1

self.W2 -= self.learning\_rate \* dW2

self.b2 -= self.learning\_rate \* db2

def train(self, X, y):

"""Train the network"""

for \_ in range(self.epochs):

self.forward(X)

self.backward(X, y)

def predict(self, X):

"""Make predictions"""

predictions = self.forward(X)

return (predictions > 0.5).astype(int)

# XOR Dataset

X = np.array([[0, 0], [0, 1], [1, 0], [1, 1]])

y = np.array([0, 1, 1, 0])

# Train MLP

mlp = MLP(input\_size=2, hidden\_size=2, output\_size=1, learning\_rate=0.5, epochs=10000)

mlp.train(X, y)

# Test MLP

print("\nMLP Predictions for XOR:")

for i, inputs in enumerate(X):

prediction = mlp.predict(inputs.reshape(1, -1))

print(f"Input: {inputs}, Predicted Output: {prediction[0][0]}")

# Visualization of Decision Boundary

def plot\_decision\_boundary(model, X, y):

x\_min, x\_max = X[:, 0].min() - 0.5, X[:, 0].max() + 0.5

y\_min, y\_max = X[:, 1].min() - 0.5, X[:, 1].max() + 0.5

xx, yy = np.meshgrid(np.linspace(x\_min, x\_max, 100), np.linspace(y\_min, y\_max, 100))

# Predict for each point in the meshgrid

Z = np.array([model.predict(np.array([[a, b]])) for a, b in zip(xx.ravel(), yy.ravel())])

Z = Z.reshape(xx.shape)

plt.contourf(xx, yy, Z, alpha=0.3, cmap=plt.cm.Paired)

plt.scatter(X[:, 0], X[:, 1], c=y, edgecolors="k", cmap=plt.cm.Paired, marker="o", s=100)

plt.xlabel("X1")

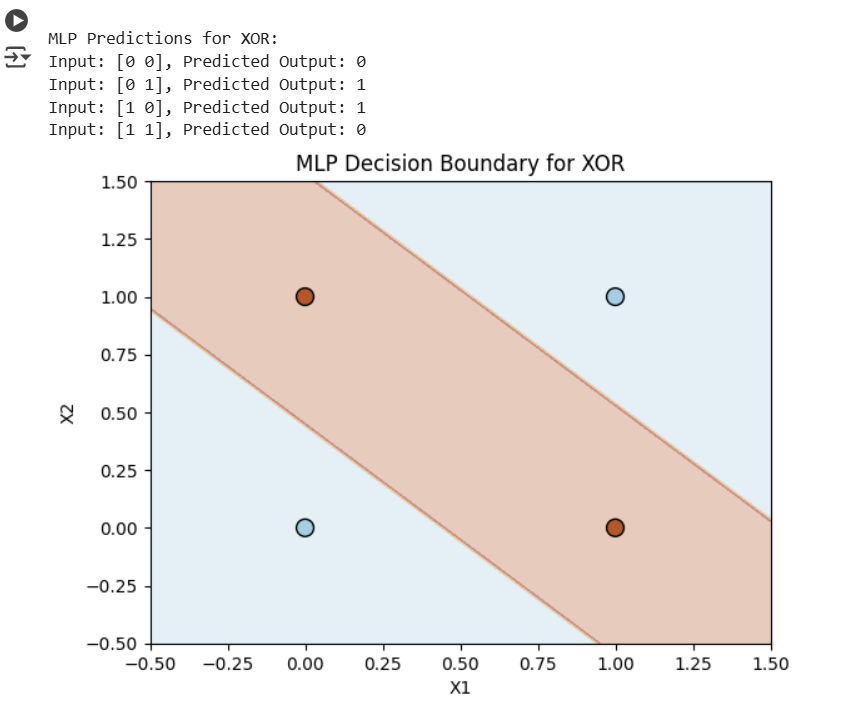
plt.ylabel("X2")

plt.title("MLP Decision Boundary for XOR")

plt.show()

# Plot decision boundary

plot\_decision\_boundary(mlp, X, y)



Note: The decision boundary visualization shows how the hidden layer transforms the input space, creating a non-linear separation for XOR.